Simulated Car Rental System

Original Requirements:

- *Design and implement a simulated Car Rental system* using object-oriented principles.
- Recommended time for this task is 2-4 hours.
- You can use any language you prefer for this exercise.
- The system should allow reservation of a car of a given type at a desired date and time for a given number of days.
- There are 3 types of cars (sedan, SUV and van).
- The number of cars of each type is limited.
- Use unit tests to prove the system satisfies the requirements.
- Please be prepared to discuss the design and implementation during the interview.

Functional Requirements (FR)

*These describe what **the system should do.***

- **User & Account Management**
  - Users can **register** with a username and password.
  - Users can **log in** securely.
  - Users can **view and update** their account information.
  - Admins can **manage user accounts** (view, update, delete).
- **Vehicle Management**
  - Admins can **add, update, and remove vehicles**.
  - Vehicles have types: **Sedan, SUV, Van**.
  - Each vehicle has limited availability (inventory).

➢ **Search & Availability**

- Users can **search for vehicles by type**.
- Users can **check availability** for a given date and time for a number of days.
- The system must prevent **overbooking**.

➢ **Reservation Management**

- Users can **reserve a vehicle** for a specified period.
- Users can **modify or cancel** a reservation before the start date/time.
- Users receive a **reservation confirmation** with details.

- **Rental Agreement**
  - The system generates a **rental agreement** for each reservation.
  - The agreement includes vehicle info, reservation dates, user info, and duration.
- **Reporting**
  - Admins can **view reservations** for vehicles.
  - Admins can **track vehicle usage and availability**.

➢ **Unit Testing**

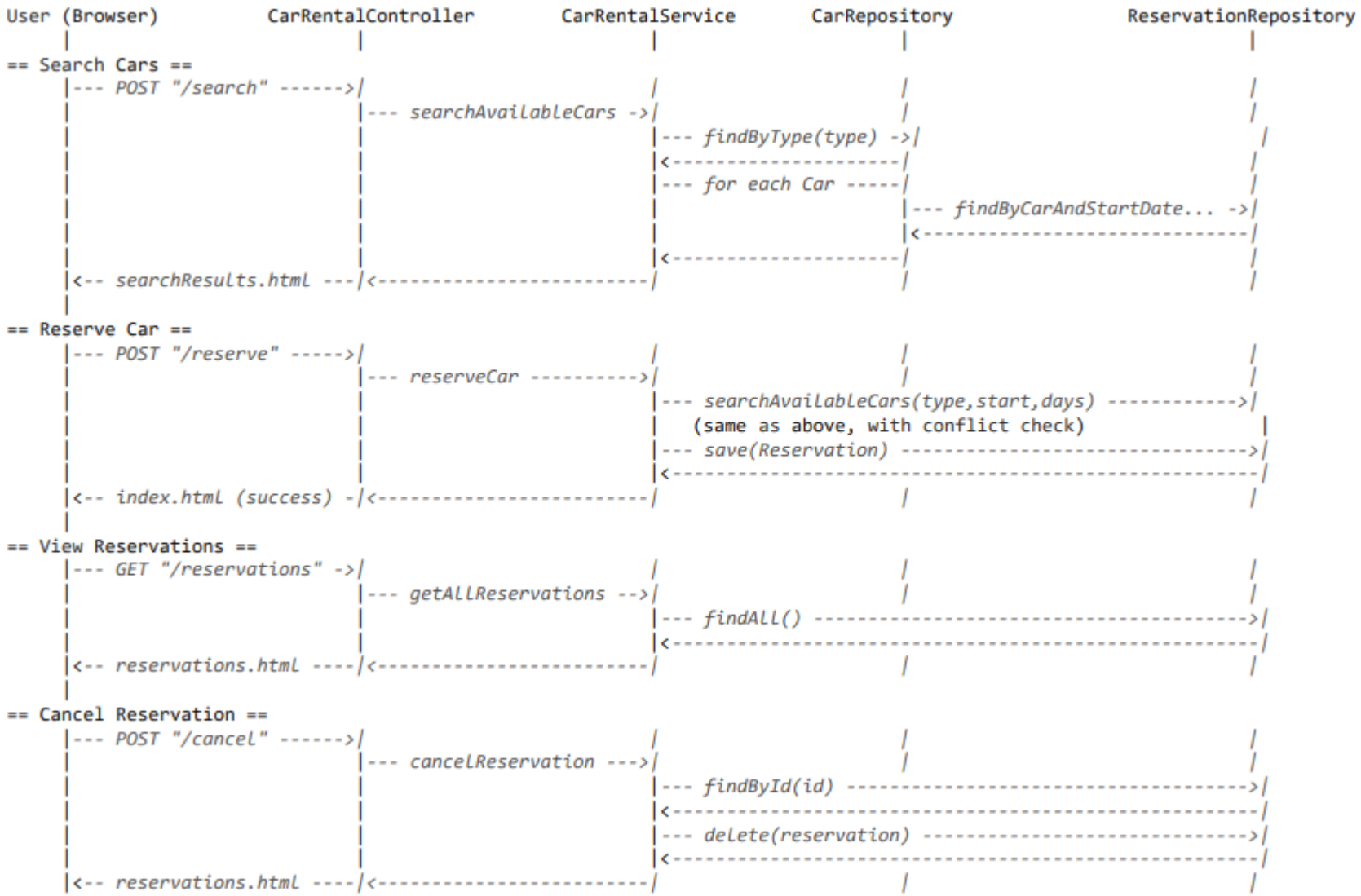
- The system must include **unit tests** to validate core functionalities, such as reservation, availability, and inventory limits.

Non-Functional Requirements (NFR)

*These describe **how the system performs** rather than what it does.*

- **Performance**
  - The system should return search results in **under 2 seconds** for standard inventory sizes.
- **Scalability**
  - The system should handle **multiple simultaneous reservations** without errors.
- **Reliability**
  - Reservations must be **accurate** and prevent double-booking.
  - Data consistency must be maintained across concurrent operations.
- **Security**
  - Passwords must be **encrypted**.
  - Only authorized users can access admin functions.
- **Usability**
  - The UI should be **intuitive**, allowing users to reserve cars quickly.
  - Error messages should be **clear** when a reservation fails.
- **Maintainability**
  - Code should follow **modular design**, making it easy to update vehicle types or add features.
- **Portability**
  - The system should run on standard platforms (Windows, Linux) with **minimal configuration**.
- **Availability**
  - System should aim for **high uptime**, e.g., 99% availability during business hours.

Sequence Diagram



Design Patterns Used in This Project

- ***Model View Controller (MVC):*** CarRentalController (Controller) , index.html, reservations.html, searchResults.html (View), Entities like Car, Reservation, User (Model)
  - ✓ **Why:** Separates request handling, UI rendering, and business logic.

- ***Repository Pattern:*** CarRepository and ReservationRepository (interfaces extending JpaRepository)
  - ✓ **Why:** Hides database details behind an interface. You just call findAll(), save(), etc. without worrying about SQL.
- ***Service Layer / Facade Pattern:*** CarRentalService is acting like a **Facade** that hides complexity of repositories and business rules.
  - ✓ **Why:** Provides a single-entry point for car rental operations (reserve, search, list reservations).
- ***Dependency Injection (DI) / Inversion of Control (IoC):***CarRentalController → injects CarRentalService, CarRentalService → injects CarRepository and ReservationRepository
  - ✓ **Why:** Instead of creating objects manually, Spring injects them at runtime. This is basically the **Strategy pattern** + IoC combined.
- ***Singleton Pattern:*** Spring beans like CarRentalService, CarRentalController are singletons by default.
  - ✓ **Why:** Ensures one shared instance per Spring context.
- ***Factory Method (via JPA):*** Spring Data JPA generates implementations for CarRepository and ReservationRepository at runtime.
  - ✓ **Why:** You don’t instantiate them yourself; Spring uses a factory to provide the correct implementation.

Object Oriented Features

- **Encapsulation** → private fields + getters/setters
- **Abstraction** → controllers vs services
- **Polymorphism** → interfaces & Spring dependency injection
- **Composition** → Reservation has Car
- **Inheritance** is minimal now, but easily extendable